Fractals Unraveled

by Jeff Stefan

This article explains the concepts of fractal geometry and provides you with a simple program that helps you to generate your own fractal images. When I first saw fractal images, I was struck by their jagged, colorful electric forms. The images appeared to be terribly complicated, but really only required a few lines of code to produce. I had an executable file (with no source code) that generated fractal images, but I never took the time to really understand how they were created and the underlying mathematical concepts behind them. Unraveling fractals comes down to understanding two concepts: iterative functions and complex arithmetic.

What are fractals?

We experience life in four dimensions: length, width, height, and time. Our senses tell us this. The irreversibility of events gives us our intuition of time. Euclidean and analytic geometry provide the mechanisms to model spatial relationships in the real world. It allows us to create useful abstractions of reality, but clean right angles and perfect arcs rarely occur in the natural world.

Objects in the natural world are contradictory, complex, and difficult to describe (especially mathematically), yet appear to be governed by a recurring set of simple underlying principles.

Tree bark looks the same from a distance, but each individual piece is somewhat different. Every ocean wave breaking against a rocky shore is a little different, even if it contains the same volume of water and travels at the same velocity. Look at the clouds and they all have similar shapes, but all are a little bit different, especially around the edges.

This is called self-similarity, a phrase coined by Benoit Mandelbrot, the father of the Mandelbrot Set. A fractal object looks almost the same from a great distance or on a microscopic scale. Fractal objects appear to be similar on all scales, but not exactly alike.

Furthermore, the dimensions of these objects do not clearly seem to be separated into one, two, or three dimensions, but somewhere in between. Hence, the word fractal for fractional dimension (also coined by Mandelbrot).

Iterative Functions

To begin learning how to generate fractal images, we first start with iterative functions. An iterative function calculates a value, then uses the value as the basis to calculate the next value. This process is repeated indefinitely. An iterative function, $Z = Z^2 - 1$, is shown in Figure 1.

In Figure 1, the starting value of Z is two. Notice how quickly the resulting Z values grow in magnitude. We can safely assume that Z will continue to grow toward infinity.

Figure 1

Initial Z = 2

$Z = Z^2 - 1$ *Resulting Z = 3*

$Z = Z^2 - 1$ *Resulting Z = 8*

$Z = Z^2 - 1$ *Resulting Z =63*

$Z = Z^2 - 1$ *Resulting Z = 3968*

We can track the progress of each step in the equation graphically, and can consider each iteration of the equation as a new state. The transitions between these new states are depicted by arrows drawn between the nodes, with the arrowhead indicating the progress of the numbers as they iterate through the equation.

This is called a state machine. Figure 2 interprets each new number generated as a new state in a state machine. This state machine knows no bound and heads straight for positive infinity when the initial value of two is used.

For the starting value of two, this equation has infinity as an attractor. If we use the same equation with a starting value of Z less than zero, the resulting numbers rapidly shrink toward zero, as illustrated in Figure 3. For a starting value of one or less, the equation oscillates between attractors of zero and negative one.

Figure 4 shows the state machines for the equation $Z = Z^2 - 1$ for the starting value of Z = .9.

For the initial value of one, Z immediately orbits around negative one and zero. If the starting value is reduced to .9, then it takes a little longer for Z to settle down to and orbit around negative one and zero. It's easy to see by the state machine that the numbers rapidly start approaching negative one and zero with each iteration of the equation.

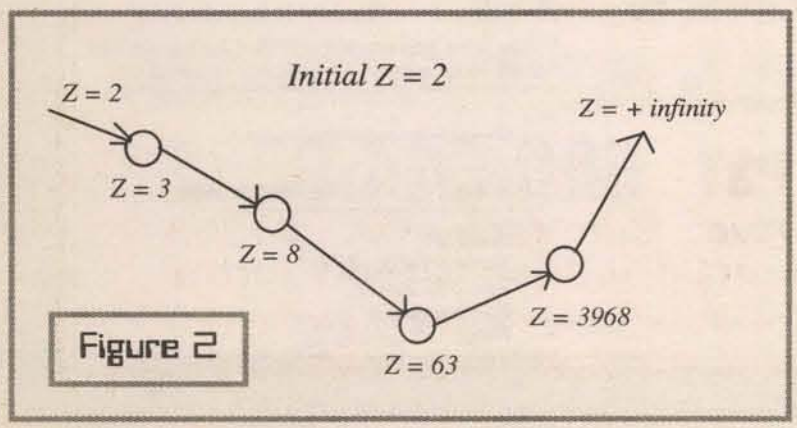


Figure 2

This may seem dull and obvious for real numbers, but it explodes in chaotic complexity and detail when complex numbers are used instead. The key points of understanding here are orbits and attractors.

Complex Numbers: A Quick Review

The hurdle in generating fractals is understanding complex numbers. Complex numbers are composed of two parts: a real part and an imaginary part. The imaginary part is defined as the square root of negative one. The square root of negative one is given the universal symbol *i*. By definition, i^2 is equal to negative one.

Any real number can be expressed as a complex number. For example, the number 2 can be expressed as $2 + 0\ i$. Zero times *i* is of course, zero.

Complex addition is simple, as shown in Figure 5. You just add like terms.

Complex multiplication is slightly more complicated, as shown in Figure 6. The trick is to pay attention to the fact that i^2 is equal to negative one.

When graphing a complex equation, the real part of the expression is plotted along the X axis, and the complex part is plotted along the Y axis.

The equation we're interested in is $Z = Z^2 + C$. Figure 7 shows this equation when complex numbers are used instead of real numbers.

The concepts behind fractals have been around since the beginning of the 20th century. In 1905, the year Jules Verne died, a French mathematician name Pierre Fatou studied the simple equation $Z = Z^2 + C$, varying the value of Z while keeping C constant. This is trivial and unproductive for real numbers, but Fatou used complex numbers for Z and C.

He soon discovered that infinity is the equation attractor if the absolute value of Z ever grows larger than two. This is the equation used to generate the Mandelbrot set, which our program generates.

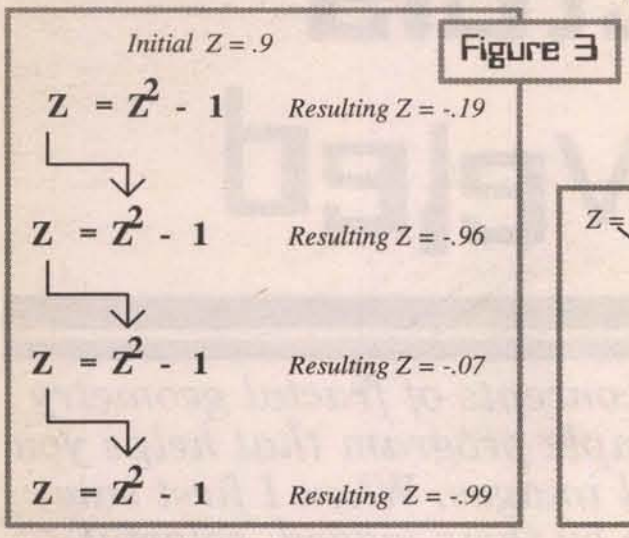
The Mandelbrot Set is generated by varying the value of C and keeping Z constant. The Mandelbrot Set is the set of numbers that do not exceed +2 or -2 when varying C in $Z = Z^2 + C$.

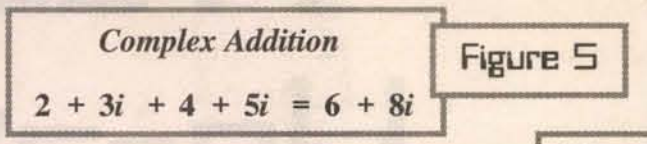
Mandelbrot, a contemporary mathematician, built on Fatou's work, studying the profound differences between analytic/Euclidean geometry and the shape of things in the real world. In 1975, he coined the word fractal, which appeared in the title of his 1977 book, *The Fractal Geometry of Nature.* The key value of the Mandelbrot Set, and fractals

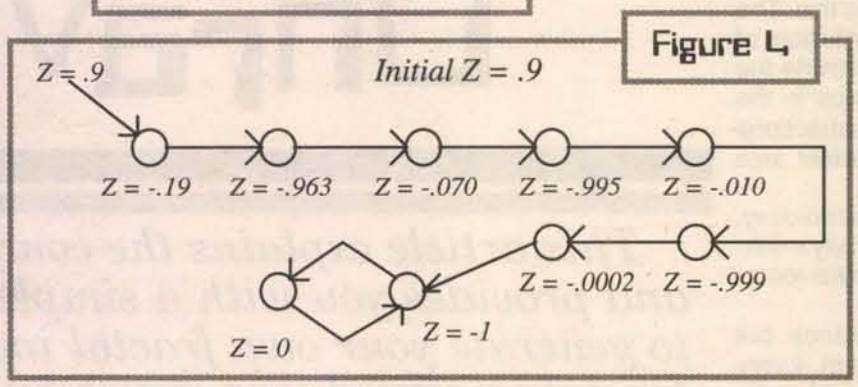
in general, is that tremendously complicated systems like those found in nature can be created from very simple rules.

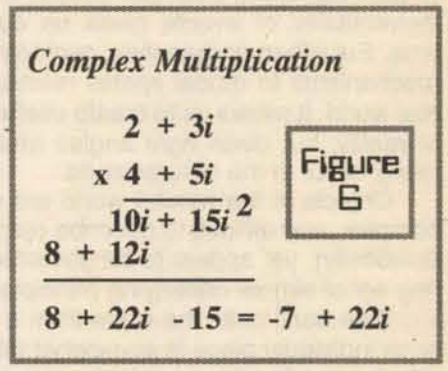
Figure 8 illustrates the full Mandelbrot Set generated from our program.

Figure 9 shows the wild effects of slight magnification and slightly different starting values.









Code Description

The program to generate the Mandelbrot Set is called mbs.c. It's written in Turbo C/C++ Version 3.0 and uses the Borland Graphics Interface (BGI) for graphics support. The file egavga.bgi must be in the same directory as mbs.exe or the program

```
/*************************************************************
* Filename: mbs.c
* Description: Mandelbrot Set Generator
*          This program is written using Borland C/C++ ver. 3.1
*          or Turbo C/C++ ver. 3.0 utilizing the default BGI
*          interface. The graphics runtime module egavga.bgi is
*          required to be in the same directory as this executable.
*
*  Application Notes:
*
*    1) The screen width and aspect ratio can be altered by
*       changing
*       the values of X_PIXELS and Y_PIXELS. The count limit
*       can be altered by changing MAX_COUNT. The display
*       centering can be adjusted by changing X_ADJUST and
*       Y_ADJUST.
*
*    2) The switch statement in GetColor() can be expanded and
*       enhanced to change pixel color and color resolution
*       depending on the value of the count.
*
*    3) The program can be terminated at any time by pressing a
*       key.
*
*  (c)  1997 Jeff Stefan
*
*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include <conio.h>

/*************************
* Define display parameters
*************************/
#define MAX_COUNT 50
#define X_PIXELS  450
#define Y_PIXELS  350
#define X_ADJUST  25
#define Y_ADJUST  75

/*********************
* Input file pointer
*********************/
FILE *fp;

/***********************
* Variable declarations
***********************/
int FromFile;
char TextBuffer[512];
int Count,Color,i;
int MaxCount = 0;

/***********************
* Function prototypes
***********************/
int GetColor(void);

#define DEBUG

main()
{
int j,k;
double XStart,YStart,Ca,Cb,Zx,Zy,Xtemp,Size,Result;
int gdriver = DETECT, gmode, errorcode;
int Done = 0;
int InChar;

  clrscr();
  printf("Read data from File or Keyboard?\n1 File  2 Keyboard\n");
  scanf("%d",&FromFile);

  /*************************/
  /* graphics initialization */
  /*************************/
  initgraph(&gdriver, &gmode, "");
  errorcode = graphresult();
  if(errorcode != grOk)
  {
    printf("Graphics error: %s\n", grapherrormsg(errorcode));
    printf("Press any key to halt:");
    getch();
    exit(1);
  }

  /***********************
  * initialize variables
  ***********************/
  Ca = Cb = XStart = YStart = Xtemp = 0.0;

  /*********************************
  * Get data from file or keyboard
  *********************************/
  switch(FromFile)
  {
  case 0:
  case 1:
    fp = fopen("mbs.dat","r");
    if(fp == NULL)
      exit(0);
    break;

  default:
    break;
  }

  while(!Done)
  {
    /*************************************
    * Re-init graphics to refresh screen
    *************************************/
    initgraph(&gdriver, &gmode, "");
    errorcode = graphresult();
    if(errorcode != grOk)
    {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Press any key to halt:");
      getch();
      exit(1);
    }

    /************************/
    /* get input parameters */
    /************************/
    switch(FromFile)
    {
    case 0:
    case 1:

      if(fp!=NULL)
      {
        fscanf(fp,"%lf %lf %lf
%d",&XStart,&YStart,&Size,&MaxCount);
      }
      break;

    /*****************************
    * Get input data from keyboard
    *****************************/
    case 2:
    default:
      ClearPrompt();
      printf("Enter XStart : ");
      scanf("%lf",&XStart);
      ClearPrompt();
      printf("Enter YStart : ");
      scanf("%lf",&YStart);
      ClearPrompt();
      printf("Enter size : ");
      scanf("%lf",&Size);
      ClearPrompt();
      printf("Enter Max Iterations: ");
      scanf("%d",&MaxCount);
      break;
    }
    /************************************************
    * Default to 100 if count is entered incorrectly
    ************************************************/
    if(MaxCount <= 0)
    {
      MaxCount = 100;
    }

    /************************/
    /* Create Mandelbrot Set */
    /************************/
    sprintf(TextBuffer,"XStart = %2.3lf YStart = %2.3lf Size = %2.3lf
        MaxCount = %d",XStart,YStart,Size,MaxCount);
    outtextxy(20,50,TextBuffer);

    for(j=1;j<X_PIXELS;j++)
    {
      /*************************
      * break out if keyboard hit
      *************************/
      if(kbhit())
        {
          closegraph();
          exit(1);
        break;
        }
      /*****************************************************
      * This section does the actual work by first assigning
      * values to the complex components of C, which are
      * Ca and Cb. In order to adjust the size of the
      * display, change the values of X_PIXELS and Y_PIXELS.
      *****************************************************/
      for(k=1;k<Y_PIXELS;k++)
      {
        Count=0;
        Ca = XStart+j*Size/X_PIXELS;
        Cb = YStart+k*Size/Y_PIXELS;
        Zx=Zy=0;
        /*****************************************************
        *  Stay in the loop and test until MaxCount is reached
        * or the result exceeds 4.0 (definitely not in the set).
        *****************************************************/
        do {
          /*****************************************
          *          2
          * Evaluate Z = Z + C in the complex plane
          *****************************************/
            Count+=1;
            Xtemp = Zx*Zx - Zy*Zy;
            Zy = 2*Zx*Zy+Cb;
            Zx = Xtemp+Ca;
            Result = Zx*Zx + Zy*Zy;
        } while ((Count<=MaxCount) && (Result <= 4.0));
```

$$z = z^2 + c$$

$z = x + yi$ *Complex substitition for z*

$z^2 = (x + yi)^2$ *Complex substitution for* z^2

$c = a + bi$ *Complex substitution for c*

$z = (x + yi)^2 + a + bi$ *Resulting experssion*

with

$(x + yi)^2 = x^2 + 2xyi - y^2$

Final expanded expression

$z = x^2 - y^2 + 2xyi + a + bi$

Figure 7

won't run.

You can enter set generation coordinates from a file, called mbs.dat, or directly from the keyboard. The parameters entered are the starting X and starting Y coordinates, the size or magnification of the image, and the number of iterations to determine whether a pixel is within the set or not in the set. We need to put a cap on the number of iterations, otherwise it could take a very long time for a decision to be made for each pixel.

The screen width and aspect ratio can be altered by changing the values of X_PIXELS and Y_PIXELS. The count limit can be altered by changing MAX_COUNT, and the display centering can be adjusted by changing X_ADJUST and Y_ADJUST.

Here's the loop for the set generator equation $Z = Z^2 + C$. This section of the program does the actual work by first assigning values to the complex components of C, which are Ca and Cb.

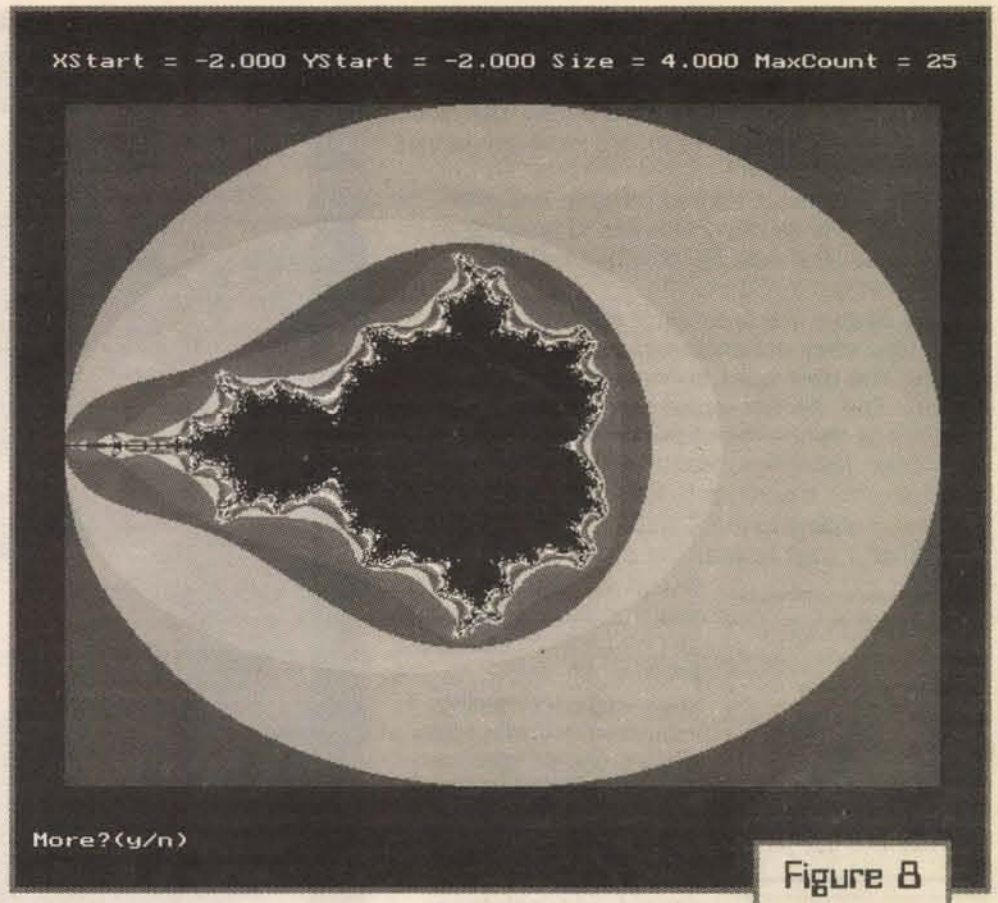


Figure 8

In order to adjust the size of the display, change the values of X_PIXELS and Y_PIXELS.

```
Count = 0;
Ca = Xstart + j *
Size/X_PIXELS;
Cb = Ystart + k *
Size/Y_PIXELS;
Zx = Zy = 0;
```

Now we need to stay in the loop until +2 or -2 is encountered, or we hit the MAX_COUNT limit value.

```
do {
    Count+=1;
    Xtemp = Zx*Zx - Zy*Zy;
    Zy = 2*Zx*Zy+Cb;
    Zx = Xtemp+Ca;
    Result = Zx*Zx + Zy*Zy;
} while ((Count<=MaxCount) && (Result <= 4.0));
```

Once we're out of the loop, we determine the color and plot it on the display:

```
    /************************************/
    /* determine pixel color and display */
    /************************************/
    Color = GetColor();
    putpixel(j+X_ADJUST,k+Y_ADJUST,Color);
   }
  }
  /*******************************
   *  Terminate program if desired
   *******************************/
  InChar = NULL;
  outtextxy(10,450,"More?(y/n)");
  InChar = getch();
  if(InChar != 'y')
  {
    Done = 1;
    if(fp)
      close(fp);
  }
 }
 /*****************************
  * Restore original screen mode
  *****************************/
 closegraph();
 return(0);
}
/*************************************************/
/* GetColor: determines the pixel Color depending */
/*          on the value of count.               */
/*************************************************/
int GetColor()
{
 switch(Count)
 {
  case 1: Color = BLUE;
  break;
  case 2: Color = GREEN;
  break;
  case 3: Color = CYAN;
  break;
  case 4: Color = RED;
  break;
  case 5: Color = MAGENTA;
  break;

  case 6: Color = BROWN;
  break;
  case 7: Color = LIGHTGRAY;
  break;
  case 8: Color = DARKGRAY;
  break;
  case 9: Color = LIGHTBLUE;
  break;
  case 10: Color = LIGHTGREEN;
  break;

  case 11: Color = CYAN;
   break;
  case 12: Color = RED;
   break;
  case 13: Color = MAGENTA;
   break;
  case 14: Color = YELLOW;
   break;
  case 15: Color = WHITE;
   break;

  case 21:  Color = BLUE;
   break;
  case 22:  Color = GREEN;
   break;
  case 23:  Color = LIGHTCYAN;
   break;
  case 24:  Color = BROWN;
   break;
  case 25:  Color = LIGHTMAGENTA;
   break;

  case 31:  Color =  RED;
   break;
  case 32:  Color = GREEN;
   break;
  case 33:  Color = BLUE;
   break;
  case 34:  Color = CYAN;
   break;
  case 35:  Color = BROWN;
   break;

  case 41:  Color = LIGHTCYAN;
   break;
  case 42:  Color = LIGHTRED;
   break;
  case 50:  Color = CYAN;

  case 51:
  case 100: Color = RED;
  break;

  case 101:
  case 125: Color = MAGENTA;
  break;

  case 150:
  case 165: Color = BROWN;
  break;

  case 166:
  case 199: Color = LIGHTGRAY;
  break;

  case 200 :
  case 225 : Color = DARKGRAY;
  break;

  case 226:
  case 250: Color = LIGHTBLUE;
  break;

  case 251:
  case 275: Color = LIGHTCYAN;
  break;

  case 276:
  case 300: Color = LIGHTRED;
  break;

  default : Color = BLACK;
  break;
 }
 return(Color);
}

/************************************
 * ClearPrompt: clears text entry area
 ************************************/
int ClearPrompt()
{
 gotoxy(30,15);
 printf("                ");
 gotoxy(30,15);
 return(0);
}
```

```
Color = GetColor();
putpixel(j+X_ADJUST,k+Y_ADJUST,Color);
```

That's the real meat of the program.The rest is just support for file and keyboard input and output, and graphics support.

The data file is named mbs.dat and can contain multiple lines of parameters to produce multiple images. The data file provided with the source code is shown in Figure 10.

As always, the program is yours to experiment with. One easy enhancement is to tinker with the colors. The pixel color is determined by the count value. The switch statement in the function GetColor() can be expanded to change pixel color and color resolution. You can plot Julia sets by changing the algorithm to keep the C value constant and changing the Z value.

When I want to examine a new area, I usually give it 25 iterations. That way it produces the image fast and with a reasonable amount of detail. If the area looks interesting, I'll increase the level of detail.

MBS.DAT

```
-2.0 -2.0 4 125
-.33 .34 1 125
-.7 -.7 .7 350
-1.5 -.15 .6 425
-.744 .097 .5 100
-.744 .097 .05 100
```

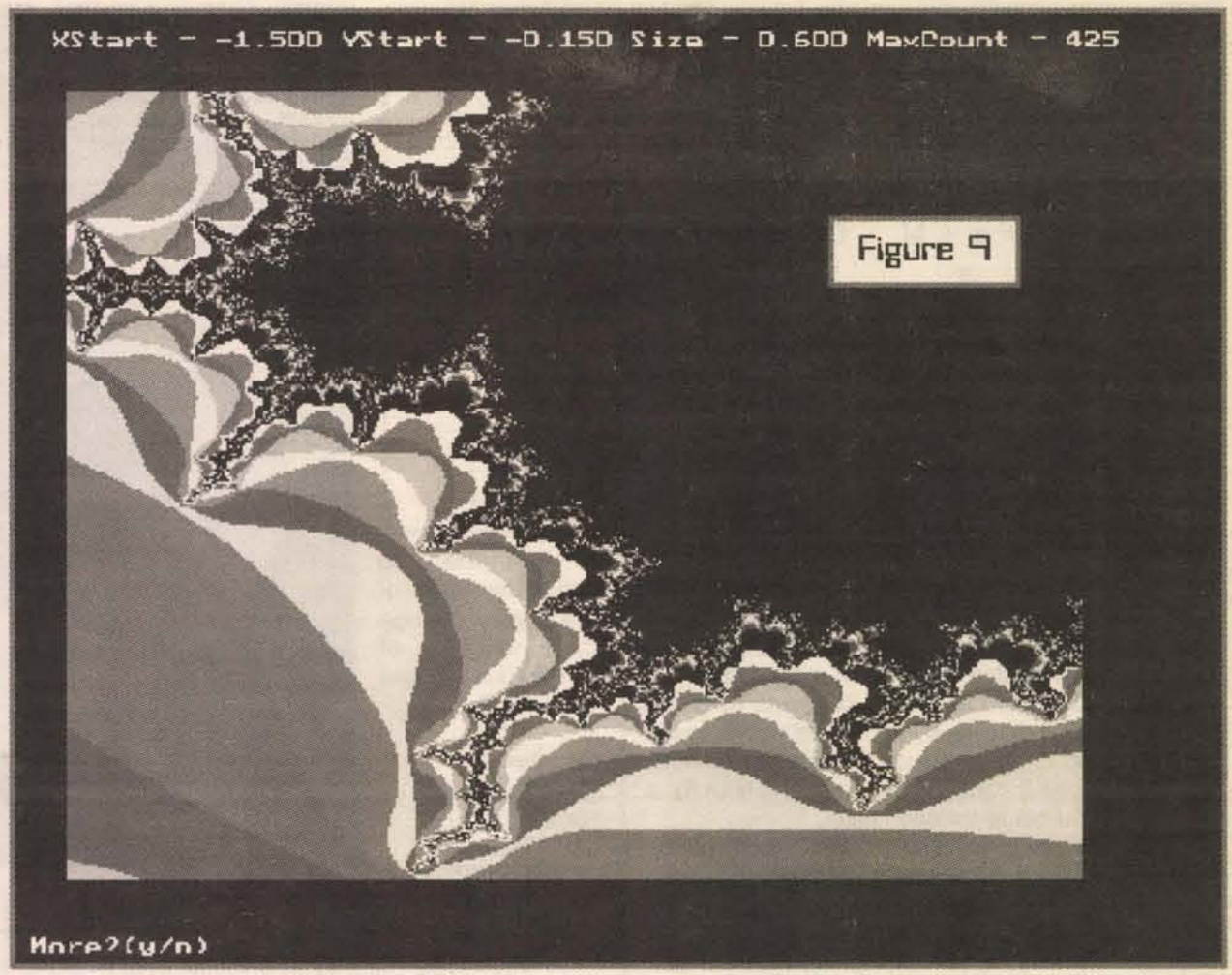
Figure 10

This program relies heavily on floating point operations, which tax the computer's horsepower. This also gives the program a degree of usefulness. I use it as a benchmark when testing a new machine's speed and performance.

The program runs exceedingly slow on a 386, reasonably well on a 486, and cruises on a Pentium. You'll get a feel for it after you run it a while on different machines. You'll be surprised at the range of performance between seemingly identical machines from different computer manufacturers. I've learned to trust it as a reliable benchmarking tool.

It's amazing that a few lines of fractal code and a personal computer are all a person needs in order to do serious mathematical research, if so inclined. Like amateur astronomy, fractal geometry opens the door for an amateur to make real and outstanding contributions to human knowledge.

Who knows what you'll discover! **NV**



Davilyn CORP.

13406 Saticoy Street, North Hollywood, CA 91605

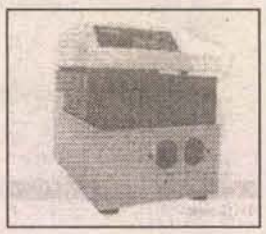
CALL FOR FREE CATALOG

818-787-3334 • 800-235-6222 • F-818-787-4732

MasterCard VISA

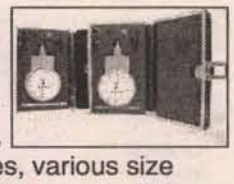
STACO VARIABLE TRANSFORMER
Input 120V, 50/60Hz.
Output 0-120/132V.
Amps 1.75 Max KVA 023.
$15

TOPAZ VOLTAGE REGULATOR
P/N 91905-68
5KVA 220/440V input
120/240V frequency
50/60 cycle.
$395



ELECTRON TUBE
JAN 12AT7
One box, 100 each.
$250

JONES MODEL 1600-1 TACHOMETER
Instrument tachometer centrifugal mechanical tach measures RPM scale 0-500 RPM, coupling shaft 1/8" accessories, various size rubber tipped drive couplings, case **new**. Also **Jones Model 1600-7** RPM scale 0-11,000 RPM w/accessories **new.** **$35/ea. or 2 for $50**



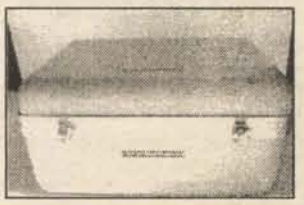
Tek 577 w/All Access, New	$2,500
Tek 7603	$495
Tek 495P Spectrum Analyzer	$10,500
HP 180A Scope	$195
HP 200CD Oscillator.	$95
HP 239A Oscillator	$650
HP 333A Distortion Analyzer	$165
HP 400EL Voltmeter, New	$350
HP X532B Waveguide Freq. Meter.	$650
HP 3478A Multimeter	$500
HP 4262A Meter	$1,750
HP 5385 Counter	$850
HP 8160A Generator	$2,500
HP 8485A Sensor.	$350
HP 8640B Signal Generator	$950
HP 86222 Plug-In	$1,000
Racal Dana 1992 Counter, New	$495
General Radio 1422D	$1,500
AN/URM-120 Wattmeter	$395

PORTABLE LANTERN
Dorco Mfg. Model 700 battery operated lantern, 4" diam., forward reflecting light and 3.25" top omnidirectional light with tilt base and handle. Uses 6V battery.
$10

SEXTANT
Aircraft or marine navigation. As-is.
$50

I.R. TANK PERISCOPE
Type M-24.
$150

FIBERGLASS BOX
Exterior size:
(L) 20-1/2" x
(W) 14-1/4" x
(H) 10-1/2".
$10



MODEL 5000 PF CAPACITOR
15KV ceramic disc.
$10

BRUNTON 2.5 HAND-HELD COMPASS
Has cast aluminum housing, precision sight. Dual bubble levels, jeweled bearings.
$65

Write in 199 on Reader Service Card.